VŠB TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Rozšíření pro ladění jazyka Monkey C v prostředí VS Code

Extension for Debugging Monkey C Language in VS Code

Ondřej Vrána

Bakalářská práce

Vedoucí práce: Ing. Jan Janoušek

Ostrava, 2021

Abstrakt

Cílem této bakalářské práce je vytvořit rozšíření pro editor Visual Studio Code. Toto rozšíření umožní ladit Connect IQ aplikace napsané v programovacím jazyce Monkey C. Mezi ladící funkce bude patřit práce s breakpointy, zkoumání proměnných, zásobníku volání a obecně vykonávání programu. Mimo ladění bude rozšíření podporovat spouštění unit testů a následný výpis výsledků.

Klíčová slova

Connect IQ, Monkey C, Visual Studio Code, Rozšíření, Ladění, Unit testy

Abstract

The goal of the bachelor's thesis is to develop an extension for code editor Visual Studio Code. The extension will allow debugging of Connect IQ applications written in the programming language Monkey C. It will offer debugging functions such as working with breakpoints, variables and call stack examination and generally program execution. The extension will also support unit tests execution and following results view.

Keywords

Connect IQ, Monkey C, Visual Studio Code, Extension, Debugging, Unit tests

Poděkování

Rád bych na tomto místě poděkoval všem, kteří mi s prací pomohli, protože bez nich by tato práce nevznikla.

Obsah

Seznam použitých zkratek a symbolů

SDK	–	Software development kit
MC	–	Monkey C
DAP	–	Debug Adapter Protocol
HTML	–	Hyper Text Markup Language
JSON	–	JavaScript Object Notation

Seznam obrázků

Kapitola 1

Úvod

Bakalářská práce je zaměřena primárně na vývoj běžně dostupných ladících funkcionalit pomocí rozšíření v prostředí editoru kódu Visual Studio Code.

Jedná se o volně šiřitelný editor vyvíjený společností Microsoft umožňujcí vysokou míru přizpůsobení a rozšíření. Všechna rozšíření lze najít na tzv. tržišti Visual Studio Code, v kterém je možné rozšíření vydat, a taktéž přímo v editoru pod kolonkou extensions. Alternativně lze rozšíření zabalit do souboru ve formátu .vsix a následně ho pomocí tohoto souboru nainstalovat přímo v editoru.

Ladění je nedílnou součástí vývoje každé aplikace, bez které nelze spolehlivě a rychle najít a odstranit všechny chyby brzdící tento proces. Jedna z dalších možností, jak otestovat určité části aplikace, jsou unit testy. Integrace unit testů do prostředí Visual Studio Code jsou také jedním z cílů této bakalářské práce mimo funkcionality ladící. Mezi další cíle patří například možnost vytvoření a vygenerování vlastního projektu a obecně zprostředkování všech funkcí, které jsou nezbytné k vývoji Connect IQ aplikace.

Kapitola 2

Connect IQ

Connect IQ je vývojářská platforma určena pro novější nositelnou elektroniku Garmin, která umožňuje jednotlivcům nebo společnostím budovat nejrůznější aplikace a rozšíření pro hodinky [1]. Tyto aplikace vylepšují a rozšiřují základní funkcionalitu těchto zařízení.

2.1 Monkey C

Monkey C je objektově orientovaný programovací jazyk vyvíjený společností Garmin Ltd. Jazyk je navržen především pro vývoj aplikací pro nositelnou elektroniku jako jsou např. chytré hodinky, fitness náramky apod. Podobá se dynamickým jazykům jako jsou Java, PHP, Ruby nebo Python. Monkey C se kompiluje do byte kódu, který je interpretován pomocí Virtuálního stroje obdobně jako Java.

Narozdíl od Javy nemá Monkey C primitivní typy - datové typy INT, FLOAT a CHAR jsou objekty. To znamená, že i primitvní typy můžou mít metody stejně jako jiné objekty. Zatímco Java využívá tzv. statického typování, Monkey C využívá tzv. kachního typování (duck typing). V jazyce Java musí vývojáři deklarovat typy pro všechny vstupní parametry funkce a typ návratové hodnoty, oproti tomu je v Monkey C využíván koncept "pokud to chodí jako kachna a káchá jako kachna, musí to být kachna".

Kompilátor jazyka Monkey C neověřuje bezpečnost typovosti a místo toho vyhazuje runtime error, pokud program zachází špatně s metodou.

Monkey C využívá moduly k shlukování tříd v aplikaci obdobně jako tomu funguje v jazyce Java pomocí jednotky balíček (package). Narozdíl od balíčku může modul obsahovat proměnné, funkce a je běžnou praxí, že statické metody existují v rámci modulu a ne nějaké určité třídy. Při importování modulu je nutné, aby všechny třídy uvnitř modulu byly referencovány jejich rodičovským modulem.

Objekty v jazycích Ruby a Python jsou hashovací tabulky a mají mnoho vlastností hashovací tabulky. Funkce a proměnné mohou být přiřazeny objektům za běhu programu. Objekty v Monkey C jsou kompilovány, tímpádem nemůžou být změněny za běhu programu. Všechny proměnné musí být

deklarovány před tím, než budou použity, a to buď v lokální funkci, instanci třídy nebo rodičovském modulu [2].

2.2 Vývojové prostředí Eclipse

Vývojové prostředí Eclipse je prezentováno společností Garmin jakožto nativní prostředí určené k vývoji aplikací na plaftormě Connect IQ. K tomu je zapotřebí mít nainstalovaný Connect IQ Eclipse plugin, který poskytuje funkcionality potřebné k takovémuto vývoji. Plugin obsahuje zvýraznění syntaxe jazyka Monkey C, integraci k sestavení programu a mimo jiné také integraci k spouštění aplikací v Connect IQ simulátoru. Connect IQ simulátor poskytuje možnost spouštění a testování aplikací předtím, nežli poběží na daném zařízení [3].

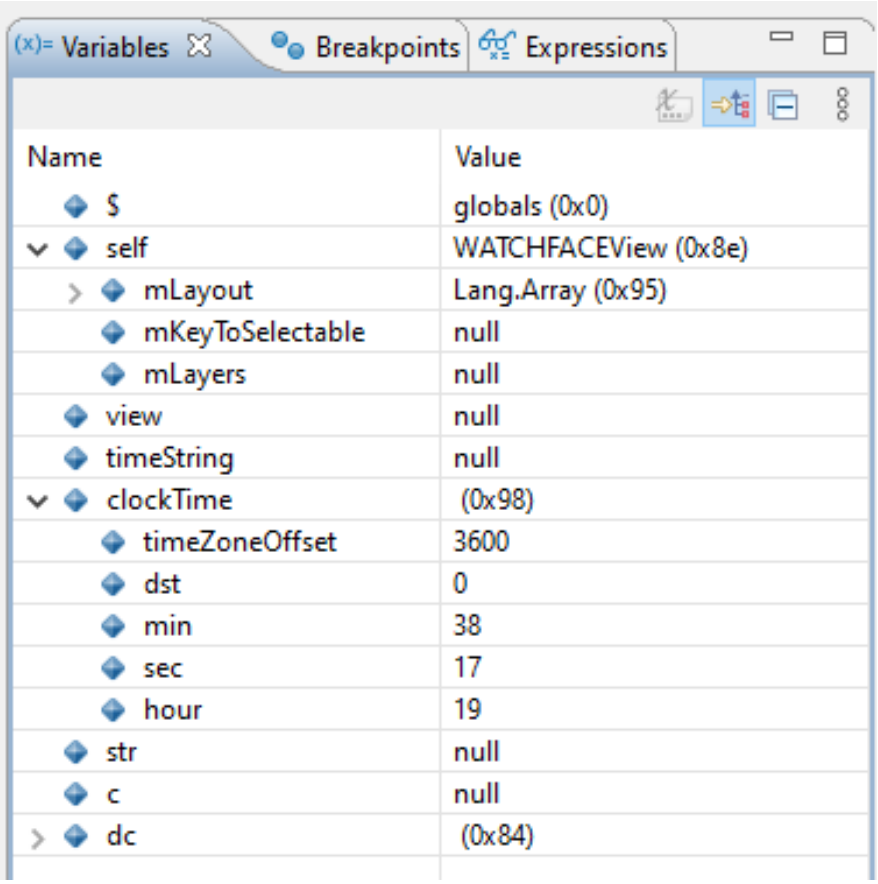
2.2.1 Ladění

Z hlediska vývoje aplikací obecně je ladění klíčovou součástí a Eclipse podporuje všechny klasické funkce, které můžeme ve vývojovém prostředí čekat. Obrázek 2.1 popisuje, jakým způsobem se nastavují breakpointy v zdrojovém kódu.

```
151
152         //clear the screen
153         targetDc.setColor(bgColor, Graphics.COLOR_TRANSPARENT);
154         targetDc.fillCircle(screenRadius, screenRadius, screenRadius + 2);
155
```

Obrázek 2.1: Nastavení breakpointu

Eclipse také poskytuje přehled všech proměnných v rámci jednoho programu viz 2.2.



Obrázek 2.2: Přehled všech proměnných

2.3 Ladící program jazyka Monkey C v příkazové řádce

V rámci vydání Connect IQ SDK verze 3.2 byl představen nový způsob, jakým lze ladit Connect IQ aplikace, a to za pomocí příkazového řádku. Ladící program je namodelován po vzoru gdb ladícího programu a nabízí klasické funkce jako jsou načítání zkompilovaných souborů, práce s breakpointy, zkoumání snímku zásobníku (stack frame), práce s proměnnými a globálním prostředím [4].

2.3.0.1 Ladění

Dokumentace všech příkazů, které je možné použít k práci s programem je dostupná po zadání příkazu `help`. Obsahuje popis příkazů, které jsou rozdělené do kategorií podle toho, k čemu slouží. Například příkaz `continue`, který slouží k pokračování ve vykonávání programu, se nechází v sekci 'running'. V popisu jsou také obsaženy parametry, které příkaz přijímá.

Před samotným načtením programu by měl být program zkompilován. To je zajištěno příkazem `monkeyc` např. s těmito parametry:

```
monkeyc -d d2deltas -f ..\\monkey.jungle -o ..\\smartarcsactive.prg -y ..\
    developer_key.der
```

, kdy ke kompilaci jsou potřeba název zařízení, cesta k .jungle souboru obsahující instrukce k sestavení, které jsou vytvořené z lokálních proměnných, hodnot, kvalifikátorů a někdy i komentářů [5]. Dále je třeba zadat cestu k vývojářskému klíči a z to z toho důvodu, že Connect IQ kompilátor vyžaduje vývojářský klíč k podepsání aplikace při kompilaci a zabalení do balíčku [3]. Před spuštěním ladícího programu je nutné mít spuštěný Connect IQ simulátor pomocí `connectiq` příkazu a následně je možné spustit ladící program pomocí `mdd`. Načtení programu do ladícího programu a současně načtení do simulátoru se provede pomocí jednoho příkazu:

```
file ..\\smartarcsactive.prg ..\\smartarcsactive.prg.debug.xml d2deltas
```

, jehož parametry obsahují cestu cestu k zkompilovanému .prg souboru a taktéž název zařízení.

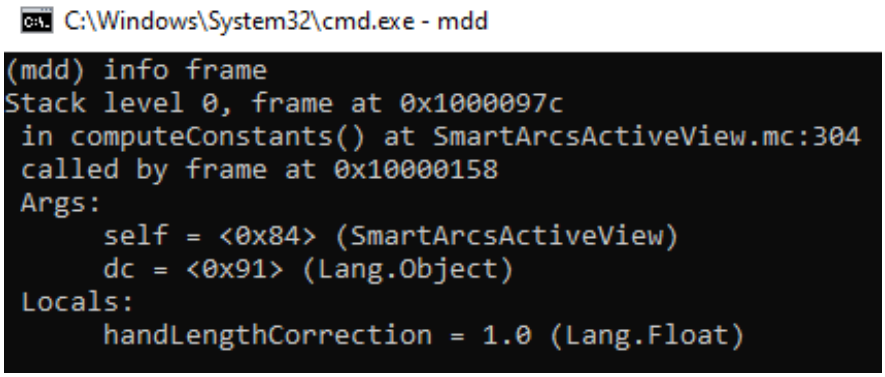
Po samotném načtení programu je možné začít zadávat příkazy na nastavení breakpointů. To se provede pomocí:

```
break SmartArcsActiveView.mc:226
```

. Název zdrojového souboru a číslo řádku, kam lze umístit breakpoint, jsou dva nezbytné parametry pro vykonání příkazu. Pro odstranění breakpointu existuje příkaz `clear` se stejnými parametry. Na deaktivaci breakpointu, tzn. aby se na místě breakpointu program nezastavil, se používá příkaz `enable` a `disable` s parametrem, který definuje rozmezí identifikačních čísel breakpointů. Breakpoint musí být umístěn na řádek, kde je možné program zastavit, v opačném případě ladící program neumožní breakpoint zadat.

Úplně prvotní spuštění programu zajišťuje příkaz `run`, po jehož zadání ladící program začne s vykonáváním programu, ale jenom do té doby, dokud nenarazí v zdrojovém souboru na breakpoint. V opačném případě program běží a jediný způsob jak aplikaci pozastavit např. z důvodu nastavení breakpointu je zmáčknutím libovolné klávesy. Program lze také krokovat řádek po řádku pomocí `next`. Pro ladící program to znamená posunutí na další řádek ve zdrojovém kódu. Příkaz `step` funguje obdobně jen s tím rozdílem, že se pokusí vstoupit do volané funkce. Jakmile se program zastaví na breakpointu, ladící program vrátí soubor a řádek, na kterém se momentálně nachází. Aby došlo k pokračování ve vykonávání programu, je nutné zadat příkaz `continue`. Program bude vykonáván do té doby než se nastaví další breakpoint nebo dojde k ukončení aplikace.

V tomto bodě programu se nabízí prozkoumat snímek zásobníku. Snímek zásobníku je kolekce všech dat na zásobníku spjatá s jedním voláním funkce, která ještě nebyla návratem ukončena. Obsahuje návratovou adresu, argumentové proměnné a lokální proměnnné viz ukázka obrázek 2.3. Pro vypsání výstupu libovolné proměnné či výrazu se používá příkaz `print`. Proměnné typu objekt, pole, slovník lze vypsat do libovolné hloubky, která se nastaví pomocí příkazu `set print max-depth X`, kde X je hloubka zanoření. Výchozí hloubka zanoření je nastavena na 1 [4].



Obrázek 2.3: Výpis zásobníku

2.4 Unit testy

Connect IQ SDK obsahuje testovací framework Run No Evil, který slouží k automatizovanému unit testování. Run No Evil operuje pouze v rámci Connect IQ simulátoru. Unit testování je způsob, jakým lze otestovat jednotlivé části programu. Každý takový test běží nezávisle a v případě, že test neprojde, je označen jako neúspěšný, a automaticky se pokračuje dalším testem.

2.4.1 Aserce

Aserce je užitečný způsob, jakým lze zkontrolovat podmínky v kritických bodech programu. Aserce se vykonávají vždy, když aplikace běží v simulátoru. Ve výpise Příklad aserce si lze všimnout, že pokud se x a y nerovnají, vypíše se definovaný výstup:

```
import Toybox.Test;

function onShow() {
  var x = 1;
  var y = 1;
  Test.assertNotEqualMessage(x, y, "x and y are equal!");
}
```

Listing 2.1: Příklad aserce

2.4.2 Příklad

Unit testy se většinou vytváří jako kterákoliv jiná třída, modul nebo funkce, přičemž musí splňovat následující požadavky [6]:

- Test metody musí být označeny anotací **:test**

- Test metody musí přijímat logger objekt jako parametr
- Test metody, které nejsou součástí testovací třídy nebo testovacího modulu musí být statické

```
class TestClass{

(:test)
static function myUnitTest(logger) {
var x = 2 + 2; logger.debug("x = " + x);
return (x == 4);
}

}
```

Listing 2.2: Ukázka testovací metody

2.4.3 Konfigurace a spouštění

Unit testy lze spustit z příkazové řádky v terminálu a jelikož Unit Testy nejsou dostupné v Eclipse pluginu, jedná se zatím o jediný způsob. Celý proces je rozdělen do 3 kroků. V prvé řadě je nutné projekt zkompilovat a to s příznakem -t. Dále je nezbytné mít zapnutý Connect IQ simulátor, čímž je umožněno spuštění `monkeydo` skriptu, který se nachází v SDK adresáři. Spuštění skriptu se provádí v příkazové řádce následovně [6]:

```
..\\monkeydo.bat ..\\WATCHFACE.prg d2bravo /t
```

Po spuštění skriptu se vykonají všechny nalezené unit testy a následně vypíšou jejich výsledky:

```
Executing test FormatTests.formatDecimalValue...
DEBUG (13:12): Actual:141.00
PASS
-------------------------------------------------------------------------------
Executing test FormatTests.formatNonDecimalValue...
DEBUG (13:12): Actual:200.51
PASS
-------------------------------------------------------------------------------
Executing test FormatTests.formatNonDecimalValue2...
DEBUG (13:12): Actual:200.50
PASS
-------------------------------------------------------------------------------
Executing test FormatTests.formatBTCPrice...
```

```
DEBUG (13:12): Actual:20000.1
PASS
```

Listing 2.3: Výsledky unit testů

Kapitola 3

Rozšíření v prostředí VS Code

Rozšíření pro prostředí VS Code jsou programy umožňující přidání nové funkcionality, témata, apod. Prostředí se dá tímto způsobem významně obohatit.

3.1 Možnosti rozšíření

Existuje několik možností, jakým způsobem lze rozšířit prostředí VS Code. Přizpůsobení vzhledu a uživatelského prostředí VS Code lze docílit změnou nebo vytvořením nového tématu. Přidání základní podpory programovacího jazyka, jako je např. automatické odřádkování nebo zvýraznění syntaxe, patří do kategorie deklarativní, což znamená, že není třeba psát žádný kód. V opačném případě funkcionality spadají do kategorie programové. Programové funkcionality přináší bohatou podporu programovacího jazyka, jako je např. automatické doplňování kódu, IntelliSense, diagnostikování errorů,... V neposlední řadě existuje aplikační rozhraní TreeView, které nabízí možnosti, jak rozšířit VS Code o nové akce v rámci průzkumníka souborů nebo vytvoření vlastního průzkumníka souborů. VS Code poskytuje základní funkcionality pro ladění programů, a proto se nabízí eventualita vytvořit rozšíření, které spojí toto ladící uživatelské rozhraní se specifickým ladícím programem [7].

3.2 Vývoj rozšíření

K vývoji VS Code rozšíření se nejčastěji používá nadmnožina programovacího jazyka JavaScript - TypeScript, do kterého se kompiluje. Oproti JavaScriptu nabízí typovost a mimo jiné třídy, moduly a rozhraní, které pomáhají ve vývoji robustních komponent [8].

VS Code aplikační rozhraní je množina všech JavaScriptových aplikačních rozhraní, které lze volat v rámci našeho rozšíření. Aplikační rozhraní je rozděleno do jmenných prostorů a tříd, kdy např. jmenný prostor s názvem commands obsahuje všechny funkce pro práci se základem celého rozšíření, příkazy. Prostřednictvím aplikačního rozhraní se dá dosáhnout přizpůsobení v oblasti

autentizace, příkazů, komentářů, ladění, správy rozšíření, jazyků, úkolů, VS Code okna a pracovní plochy [9]. Následující kód registruje příkaz pomocí funkce registerCommand a následně vypíše zprávu do informačního okna.

```
commands.registerCommand('extension.sayHello', () => {
    window.showInformationMessage('Hello World!');
});
```

Listing 3.1: Registrace příkazu

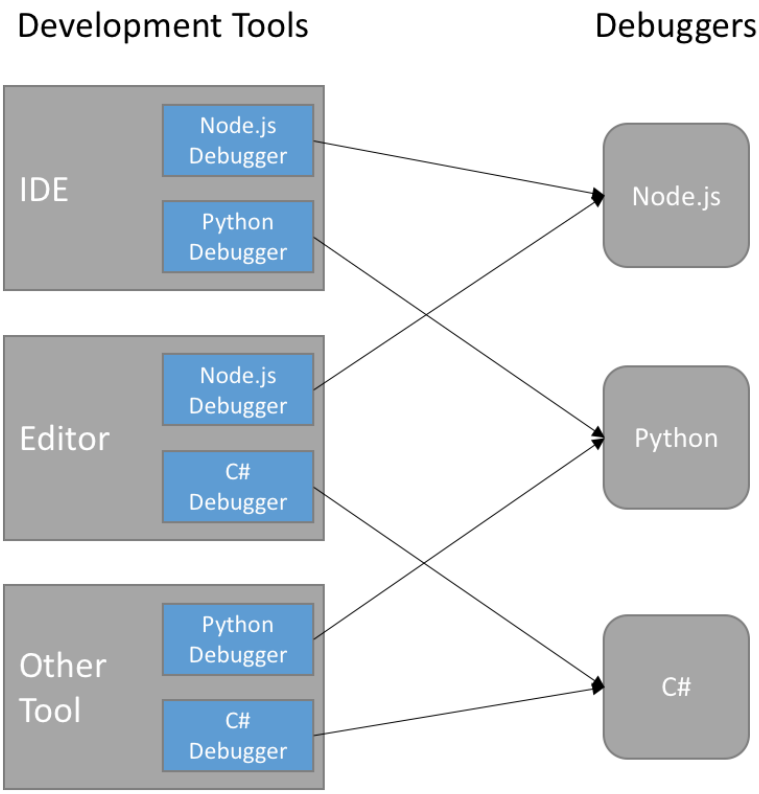
Registrace všech příkazů probíhá v metodě activate, přičemž k tomu, aby byly příkazy dostupné v rámci palety příkazů, je nutné název příkazu umístit do souboru package.json v rámci klíče activationEvents.

```
"activationEvents": [
    "onCommand:extension.monkeyc-debug.config",
    ...
    ...
    ...
  ],
```

V případě, že se do tohoto souboru neumístí, bude je možné volat maximálně programově. Kompletní dokumentaci celého aplikačního rozhraní lze nalézt zde.

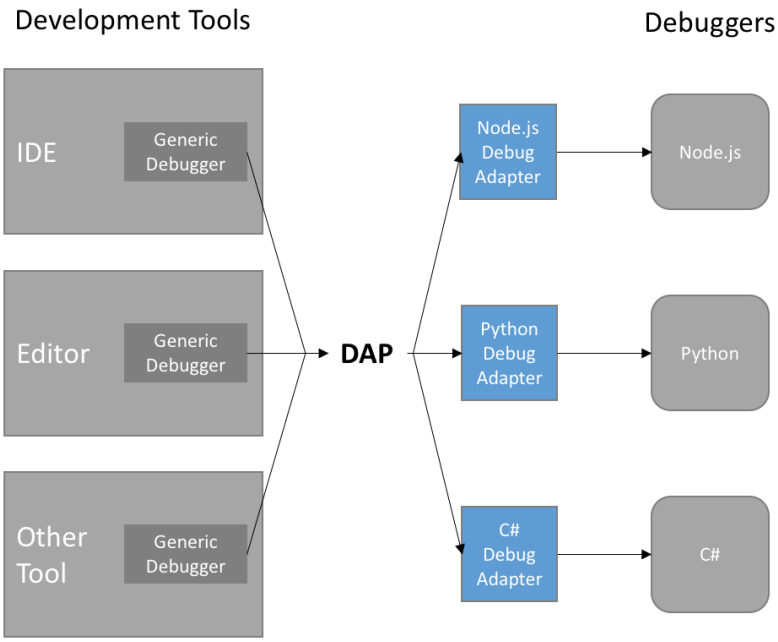
3.3 Protokol ladícího adaptéru

Protokol ladícího adaptéru (Debug Adapter Protocol) je jakýsi abstraktní protokol, který se snaží standardizovat způsob, jakým vývojářský nástroj komunikuje s určitým ladícím programem. Implementace uživatelského prostředí pro klasické ladící funkce v rámci nového ladícího programu vyžaduje značné množství úsilí a jelikož každý vývojářský nástroj používá odlišné aplikační rozhraní, dochází tedy k duplikování funkcionality, jež je visualizováno modrými boxy v obrázku 3.1:



Obrázek 3.1: Bez ladícího adaptéru

Je nerealistické předpokládat, že existující ladící programy adoptují tento protokol v blízké době a proto se zavádí zprostředkovatel, který přebírá roli adaptace protokolu existujícím ladícím programem. Z tohoto zprostředkovatele se stává tzv. **Ladící adaptér**. Výslednou architekturu popisuje obrázek 3.2.



Obrázek 3.2: S ladícím adaptérem

DAP umožňuje implementovat jedno generické rozhraní určené k ladění v rámci jednoho vývojářského nástroje a zároveň diagram ukazuje, že je možné ladící adaptér použít několikrát pro různé vývojářské nástroje.

Jakmile začne ladící relace, vývojářský nástroj potřebuje způsob, jakým bude komunikovat s Ladícím adaptérem. Komunikace není součástí specifikace protokolu, ale přesto se jedná o důležitý detail. Nabízí se dvě možnosti komunikace:

- **Režim jedné relace**: V tomto režimu se vývojářský nástroj spustí jako samostatný proces, který komunikuje pomocí stdin a stdout. Na konci relace je ladící program ukončen. Pro ladící relace, které běží současně, spouští vývojářský nástroj vícero Ladících adaptérů.
- **Režim více relací**: V tomto režimu vývojářský nástroj nespouští Ladící adaptér, ale předpokládá, že již je spuštěný a poslouchá na specifickém portu na pokusy o připojení. Pro každou ladící relaci je inicializována nová komunikační relace na specifickém portu. Na konci relace se vývojářský nástroj odpojí.

Následně začne vývojářský nástroj komunikovat s Ladícím adaptérem pomocí base protokolu.

Base protokol zprostředkovává výměnu zpráv, které se skládají z hlavičky a obsahu ve formátu JSON podobně jako je tomu u protokolu HTTP. Příklad zprávy ve formátu JSON pro žádost o posunutí na další řádek:

```
Content-Length: 119\r\n
\r\n
```

```
{
    "seq": 153,
    "type": "request",
    "command": "next",
    "arguments": {
        "threadId": 3
    }
}
```

Listing 3.2: Ukázka zprávy v rámci base protokolu

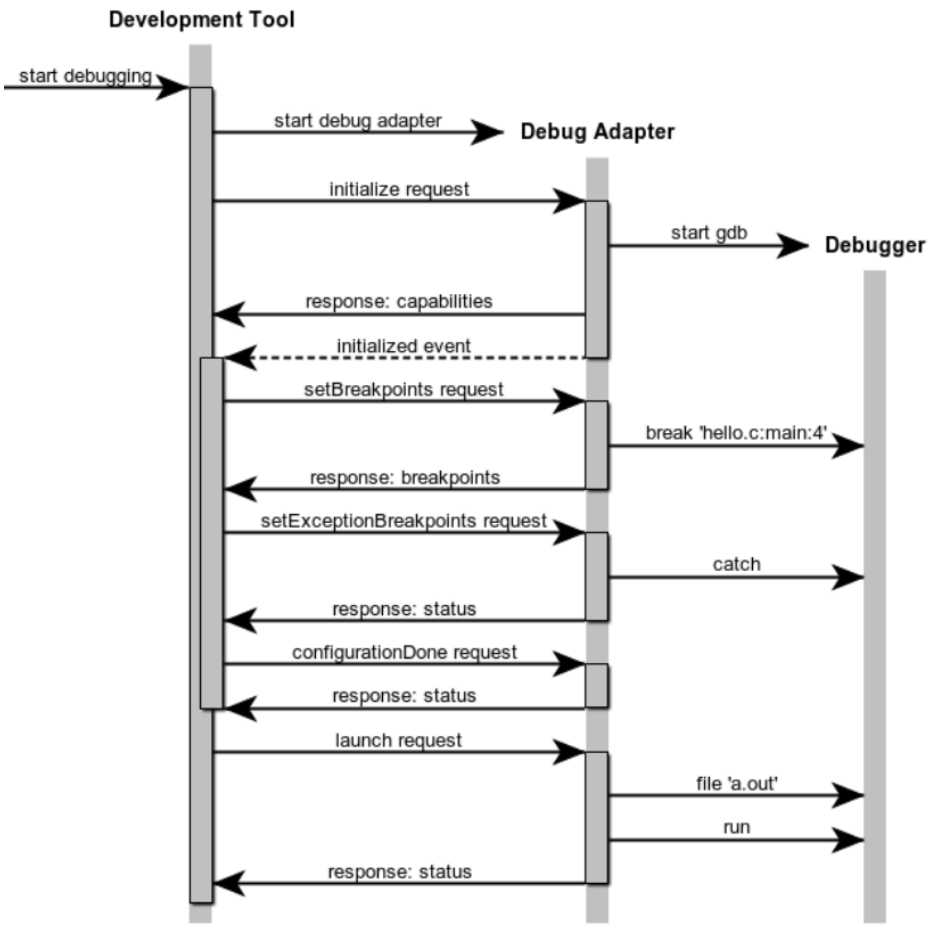
Na začátku ladící relace zašle vývojářský nástroj adaptéru žádost o inicializaci a to z důvodu výměny informace o všech vlastnostech, které ladící program podporuje.

Po úspěšné inicializaci začne Ladící adaptér přijímat žádosti o samotné spuštění ladění. V tomto bodě se nabízí dva druhy žádostí:

- **Žádost o zahájení**: Ladící adaptér spustí program v ladícím módu a poté s ním začne komunikovat. V tomto případě je Ladící adaptér zodpovědný za spuštěný program a je na Adaptéru, aby nabídl uživateli možnosti, díky kterým může s programem komunikovat. Existují tři způsoby určené k zahájení ladění: Ladící konzole: Interaktivní REPL prostředí, jež umožňuje uživateli vyhodnocovat výrazy v kontextu laděného programu. Integrovaný terminál: Terminálový emulátor integrovaný ve vývojářském nástroji, který podporuje běžný terminálový řídící kód a čtení vstupu programu, Externí terminál: Podobné integrovanému terminálu lišící se pouze tím, že integrovaný terminál běží mimo vývojářský nástroj.

- **Žádost o připojení**: Ladící adaptér se připojí k již spuštěnému programu. Uživatel je zodpovědný za spuštění a ukončení tohoto programu.

Ještě před samotným spuštěním programu dochází ke konfiguraci breakpointů a chování výjimek. Ladící adaptér zašle událost vývojářskému nástroji, která značí úspěšnou inicializaci, a tím říká, že je připravený příjímat konfigurační žádosti. Vývojářský nástroj odpoví na tuto událost zasláním žádostí o nastavení breakpointů a v případě, že podporuje i některou z těchto funkcí: breakpointy ve funkci, zastavení na výjimce, konfigurace dokončena, zašle jimi odpovídající žádosti. V následujícím diagramu 3.3 je sumarizována sekvence žádostí a událostí pro hypotetický gdb Ladící adaptér:



Obrázek 3.3: Sekvence žádostí a událostí

Pokaždé, když se program zastaví: na začátku programu, breakpointu, výjimce nebo pokud uživatel zažádal o zastavení, pošle ladící adaptér událost o zastavení programu s příslušným důvodem a identifikátorem vlákna. Ihned po přijetí události si vývojářský nástroj vyžádá vlákna a seznam snímků zásobníku (stack trace) vlákna zmíněného v události. Pokud se uživatel rozhodne prozkoumat snímek zásobníku, vývojářský nástroj nejpve pošle žádost o rozsah snímku zásobníku a následuje žádost o získání proměnných v rozsahu snímku zásobníku. Pokud je proměnná strukturovaná, tzn. obsahuje dále vnořené proměnné, zašlou se dodatečné žádosti o tyto proměnné. To vede k výslednému vodopádu žádostí:

```
Threads
  StackTrace
    Scopes
      Variables
        ...
```

```
Variables
```

Listing 3.3: Vodopád žádostí

Na konci ladící relace musí Ladící adaptér vždy zaslat událost značící konec ladění. Sekvence událostí je při ukončení ladění trošku jiná v závislosti na tom, zda došlo ke spuštění ladění pomocí žádosti o zahájení nebo žádosti o připojení [10].

Kapitola 4

Návrh a implementace rozšíření

Rozšíření pro ladění Connect IQ aplikací vyvíjených pomocí jazyka Monkey C využívá ladícího rozhraní prostředí Visual Studio Code a zároveň integruje ladící program v příkazové řádce. Implementuje většinu ladících funkcí, které jsou dostupné v rámci ladícího rozhraní.

4.1 Motivace k vytvoření ladícího rozšíření

Přestože Eclipse poskytuje ladící funkcionalitu pro Connect IQ aplikace a jedná se o plnohodnotné vývojové prostředí, je nutné podotknout, že zorientování se v tomto prostředí může být ze začátku obtížnější, a tím může odradit spoustu vývojářů. Oproti tomu editor kódu Visual Studio Code patří mezi nejpoužívanější vývojářské nástroje za poslední dobu a v době, kdy začala práce na tomto rozšíření pro ladění Connect IQ aplikací, neexistovalo žádné oficiální rozšíření od společnosti Garmin. Tento důvod vedl k rozhodnutí vytvořit právě rozšíření pro ladění Connect IQ aplikací. V současné době již existuje oficiální rozšíření pro tento editor, jenž je v aktivním vývoji, a vize Garminu je taková, že se postupem času přejde kompletně do tohoto prostředí.

4.2 Specifikace

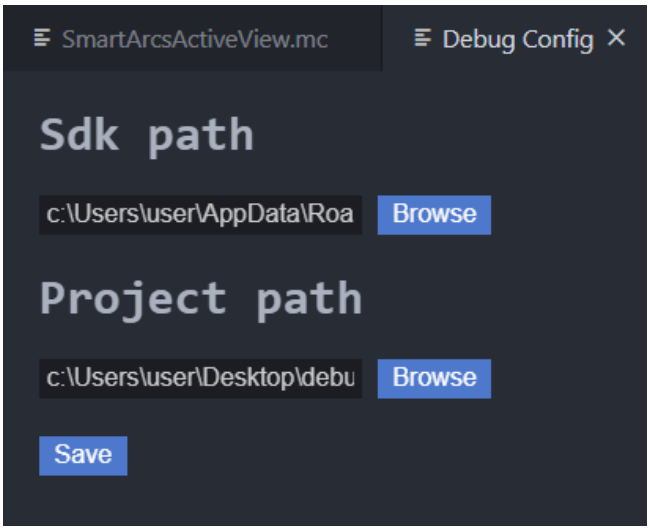
Jedná se o jednovláknový program využívající technologii Node.js, která umožňuje spouštět JavaScript mimo prostředí internetového prohlížeče [11]. Jelikož je vývoj ladícího adaptéru netriviální záležitostí, bylo využito fiktivního rozšíření od společnosti Microsoft, které není reálně napojené na žádný ladící program, a pouze simuluje ladící funkcionalitu [12]. Toto rozšíření slouží jako základ celého ladícího rozšíření.

4.3 Implementace

Jeden z počátečních problémů, který byl nutný vyřešit, byla komunikace mezi rozšířením a ladícím programem v příkazové řádce tzn. jakým způsobem se přenáší např. informace o tom, na jakém řádku se ladící program aktuálně nachází. Komunikace je základem celého rozšíření a funguje tím způsobem, že na začátku každé ladící relace se spustí příkazová řádka, která příjímá ladící příkazy jako svůj vstup a zároveň vypisuje výstup těchto příkazů na standardní výstup. Příkazová řádka se spouští jako podproces pomocí potomka (child process). Potomek poskytuje rozhraní k spouštění podprocesů v rámci Node.js programu a je několik možností, jak lze takový proces spustit. V rozšíření se spouští příkazová řádka pomocí metody `spawn` a to z důvodu neustálého zasílání zpráv mezi rozšířením a příkazovou řádkou. Metoda `spawn` vytváří podproces bez toho, aniž by blokovala Node.js cyklus událostí (event loop) [13].

Všechny funkce, jenž potřebují výstupové data, zašlou požadavek ladícímu programu a poté čekají na výstup zaslaného příkazu. Toto čekání je reprezentováno objektem Promise, který představuje úspěšné dokončení dané operace [14]. Operace se eventuálně dokončí v tom okamžiku, kdy z příkazové řádky úspěšně přijde výstup daného příkazu. Výstup přichází do metody, která je registrována jako posluchač události data. V okamžiku, kdy se připojí posluchač události k datovému proudu (stream), začnou data proudit hned, jakmile jsou vygenerována. V případě, že se posluchač události nepřipojí, žádná data se negenerují [15]. Komplikace nastávají při čtení výstupu a to z toho důvodu, že data přicházejí po částech a ne po určitých celcích jako je např. řádek či sjednocený výstup jednoho příkazu. Data se tedy ukládají do pomocné vyrovnávací paměti (buffer) a v okamžiku, kdy je zřejmé, že již přišel celý výstup jednoho příkazu, přepošle se metodě, která na něj čeká. Každá taková metoda si výstup dále zpracuje a získá informace, které z výstupu potřebuje. Všechny metody, které čekají na výstup, jsou ukládány společně s klíčem do pomocného pole, ze kterého jsou vyjmuty až po úspěšném obdržení výstupu.

Jedna z dalších věcí, která je nezbytná pro úspěšné spuštění ladící relace je konfigurace. Aby bylo možné ladit Connect IQ aplikaci, musí uživatel zvolit cestu k vývojářské sadě nástrojů a cestu k adresáři (projektu), ve kterém je aplikace umístěna. K této konfiguraci slouží příkaz Connect IQ debug config, který lze najít v paletě příkazů. Samotné konfigurační prostředí je realizováno pomocí aplikačního rozhraní Webview. Rozhraní nabízí kompletně přizpůsobitelné zobrazení či pohled v prostředí VS Code. Pomocí tohoto rozhraní lze vytvořit komplexní uživatelské rozhraní nad rámec toho, co podporuje nativní aplikační rozhraní VS Code. Webview představuje vložený HTML dokument - iframe, který je schopný zobrazit téměř všechen obsah ve formátu HTML. Komunikace mezi Webview a rozšířením probíhá čistě pomocí zasílání zpráv.



Obrázek 4.1: Konfigurační webview

Poté, co uživatel zvolí příkaz Connect IQ debug config 4.1, se otevře v nové záložce konfigurační webview viz 4.1 a následuje vyplnění nezbytných cest v rámci formuláře pomocí souborového dialogu. Jakmile dojde k uložení formuláře, webview zašle zprávu s oběma cestami pomocí funkce `postMessage`, která je součástí speciálního objektu aplikačního rozhraní. Data přichází z webview do metody, jež je registrována jako posluchač události `onDidReceiveMessage`, a posléze jsou ukládána do konfiguračního souboru ve formátu JSON obsahující informace potřebné k zahájení ladící relace.

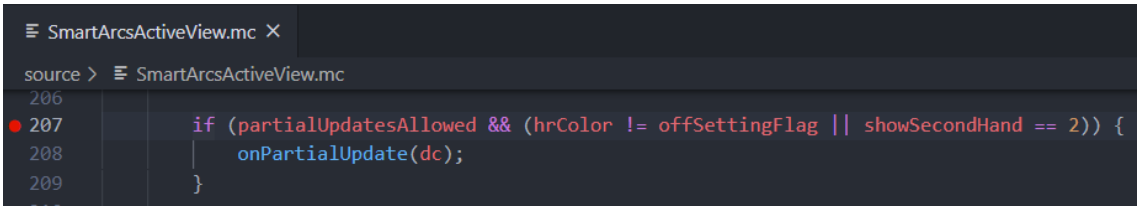
```
Získat cesty ze vstupu
Text zprávy := cesta k projektu + cesta k sdk složce
Příkaz := data
Zaslat zprávu po kliknutí na tlačítko
```

Listing 4.1: Pseudokód pro zaslání zprávy z webview směrem do rozšíření

Jelikož je výměna zpráv obousměrná, lze také poslat zprávu z rozšíření směrem do webview. Zprávy jsou doručeny jen tehdy, jeli webview aktivní nebo je otevřené na pozadí. Samotné zaslání zprávy zprostředkovává metoda `postMessage`, která je dostupná pomocí objektu webview. Aby došlo k úspěšnému odeslání zprávy do webview, je nutné znát odkaz na současný panel, jenž obsahuje cílové webview. Tento odkaz se uloží při aktivaci webview do proměnné ještě před samotným odesláním zprávy a poté se v posledním kroku zavolá příkaz sloužící k zasílání zpráv s parametrem obsahu zprávy [16]. V tomto případě reaguje webview na událost message v metodě, která je k této události registrována.

Rozšíření podporuje práci s breakpointy, to znamená přidání, odstranění, vypnutí a zastavení programu na breakpointu. V případě přidání a odstranění přijdou v požadavku o nastavení breakpointů vždy všechny breakpointy existující pro jeden zdrojový soubor tzn., že u přidání přijdou

všechny původní breakpointy v jednom zdrojovém souboru společně s přidaným breakpointem a při odstranění přijdou všechny breakpointy vyjma toho odstraněného. Mazání původních breakpointů probíhá pomocí funkce, která si načte breakpointy v cílovém souboru, a poté smaže breakpointy dle jejich jedinečných identifikátorů. Informace o breakpointech má rozšíření uložené v datové struktuře Map, jenž má data uložena ve formátu klíč-hodnota, kde klíč znázorňuje cestu k souboru a hodnota je pole objektů reprezentující breakpointy. Přídání a odstranění funguje na podobném principu a to takovém, že při každé žádosti se odstraní všechny breakpointy, které se vyskytují v daném zdrojovém souboru, a poté se nastaví všechny breakpointy, které přišly v žádosti od klienta. Rozšíření sestaví pro každý takový breakpoint příkaz, složený z cesty k souboru a číslem řádku, a zašle jej do příkazové řádky.



Obrázek 4.2: Nový breakpoint

Pro breakpoint nastavený na obrázku 4.2 se zašle tento příkaz:

```
break SmartArcsActiveView.mc:207
```

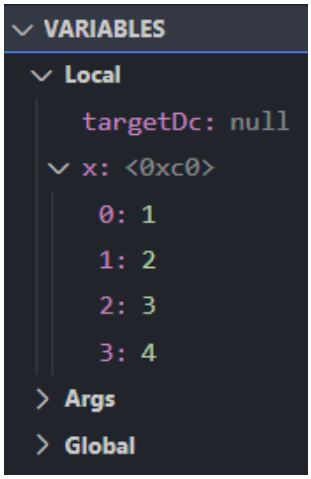
Breakpointy lze nastavovat kdykoliv před spuštěním ladící relace a během. Pokud dojde k nastavení breakpointů před ladící relací, avšak program se na žádném z nich nezastaví, znamená to, že aplikace běží bez zastavení. V tomto případě se nabízí umístit breakpoint na to místo, kde se dá očekávat, že vstoupí program. Aby bylo možné zadávat příkazy, musí ladící program pozastavit vykonávání programu. Pro opětovné spuštění programu je nutné zadat příkaz continue poté, co se dokončí zadávání příkazů. Protokol ladícího adaptéru nabízí volitelný požadavek configurationDone, jenž značí úspěšné dokončení konfigurace. Tato konfigurace mimo jiné zahrnuje nastavení všech breakpointů ve všech souborech. Komplikace nastávají z toho důvodu, že tento požadavek je dostupný pouze pro prvotní spuštění ladící relace. Pokud nastane situace, kdy je nutné po nastavení breakpointů pokračovat a nejedná se o prvotní spuštění, požadavek breakpointy zadá a následně se pokusí pokračovat zavoláním metody `continueFn`. Metoda čeká 5000 milisekund než zavolá metodu, která přímo zasílá ladícímu programu informaci o tom, že má pokračovat ve vykonávání. Pokud ovšem v tomto časovém intervalu nepřijde další požadavek a nevynuluje časový interval. V této situaci je časový interval resetován a čeká se dalších 5000 milisekund za předpokladu, že jeden požadavek netrvá více než 5000 milisekund.

```
public continueFn() {
    if (this._timer) {
```

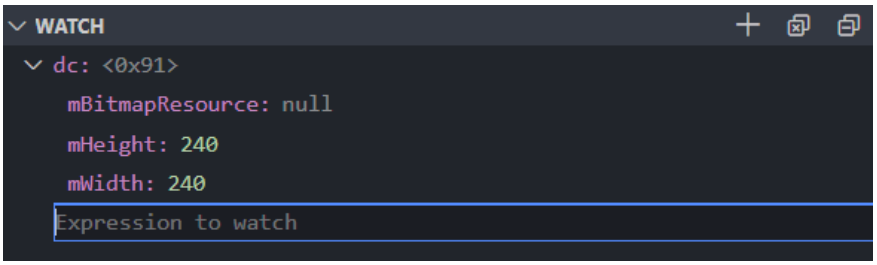
```
            clearTimeout(this._timer);
        }
        this._timer = setTimeout(async () => {
            this._timer = null;
            await this.continue();
            this.sendEvent('continuedAfterSetBreakpointsRequest');
        }, 5000);
    }
```

Listing 4.2: Metoda

V rámci Connect IQ aplikace existuje několik druhů proměnných. Lokální proměnné, jež mají platnost v oblasti funkce nebo bloku, globální proměnné, které jsou dostupné v rámci celého programu, instanční proměnné a vstupní parametry funkce. V prostředí VS Code jsou 3 možnosti, jakým způsobem lze zkoumat proměnné. Jedna z možností je využití seznamu všech dostupných proměnných viz 4.3, který se nachází po levé straně VS Code okna. Pokud se nejedná o proměnnou primitivního typu, nachází se vedle názvu proměnné šipka, po jejíž kliknutí se odkryjou dále vnořené proměnné u proměnných typu objekt, pole, slovník.



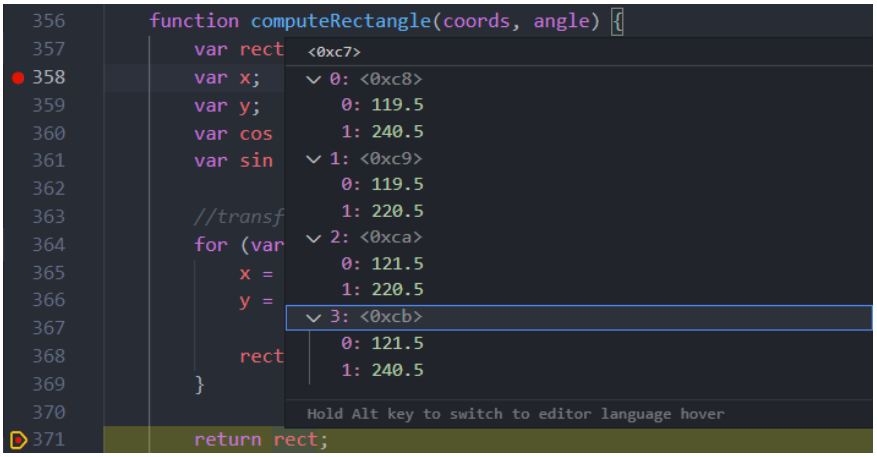
Obrázek 4.3: Výpis proměnných



Obrázek 4.4: Kukátko

Další z možností je pomocí kukátka (watch) viz 4.4. Do kukátka lze přidat jakýkoliv výraz, který se následně vyhodnotí, a jehož hodnotu lze sledovat. Tímto způsobem se dá pozorovat např. jak se změnila hodnota proměnné po vykonání určité funkce.

Jako poslední možnost se nabízí najet myší přímo na proměnnou v zdrojovém kódu viz 4.5 [17].



Obrázek 4.5: Detail proměnné po najetí myší

Proměnné se načítají při každém zastavení ladícího programu na breakpointu. Lokální proměnné, instanční proměnné a parametry funkce lze získat z výstupu příkazu `info frame`. Výstup tohoto příkazu je nutné zpracovat a získat veškeré potřebné informace o proměnných. Lokální proměnné jsou sjednocenné pod slovem Locals, zatímco instanční proměnné a parametry funkce lze najít pod slovem Args s tím, že instanční proměnné jsou zahrnuty pod objektem Self viz obrázek 3. Pro výpis globálních proměnných je nutné zadat příkaz `info variables`. Všechny proměnné mají sjednocený formát výstupu, kdy za jménem následuje hodnota a za hodnotou se nachází datový typ v kulatých závorkách. Proměnné, které jsou vnořené do proměnných neprimitivního typu lze poznat z výstupu podle délky odsazení. Na výpise *Proměnná typu Lang.Object* je možné vidět proměnnou typu Lang.Object, jejíž vnořené proměnné jsou na výpise odsazeny, a nachází se na řádcích pod touto proměnnou.

```
dc = <0x91> (Lang.Object)
```

```
  mBitmapResource = null,
  mHeight = 240 (Lang.Number),
  mWidth = 240 (Lang.Number)
```

Listing 4.3: Proměnná typu Lang.Object

Výpis *Proměnná typu Lang.Dictionary* obsahuje výstup proměnné typu slovník, jenž v Monkey C reprezentuje kolekci párů klíč-hodnota.

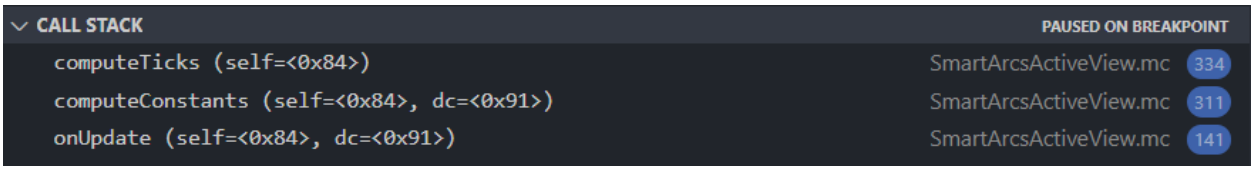
```
dict = <0x8d> (Lang.Dictionary)
  size = 2
  {
    <0x8e> (Lang.String)
      size = 1
      "c"
    =>
    1 (Lang.Number),
    <0x8f> (Lang.String)
      size = 1
      "d"
    =>
    2 (Lang.Number)
  }
```

Listing 4.4: Proměnná typu Lang.Dictionary

Aby bylo možné proměnné zobrazit v rámci VS Code rozhraní, musí být nejprve zpracovány a převedeny z textové podoby do datového typu Proměnná (Variable), kterou definuje Protokol ladícího adaptéru. Celý proces funguje tím způsobem, že poté co se ladící program zastaví na jakémkoli řádku, přijde od klienta požadavek o proměnné s parametrem, který určuje jaký druh proměnných se má načíst. V rámci tohoto požadavku se zavolá metoda, která komunikuje s ladícím programem, a zpracovává textový výstup do pole proměnných. Jelikož výstup příkazu `info frame` neobsahuje vnořenné proměnné proměnných neprimitivního typu, je nutné proiterovat vypsané proměnné řádek po řádku a pokud se nejedná o proměnnou primitivního typu, donačíst vnořené proměnné pomocí příkazu `print`. Výstup tohoto příkazu se zašle metodě, která prochází výstup řádek po řádku a rekurzivně vytváří stromovou strukturu až do maximální hloubky, která je nastavena na 100. Při každé iteraci se uloží současná velikost odsazení, která se dále předává jako parametr dalšímu volání - iteraci. Pokud je při dalším uložení odsazení stejné velikosti jako předané odsazení, uloží se proměnná do pole proměnných, které se taktéž předává jako parametr. Toto pole představuje vnořenné proměnné, které se postupně plní. V případě, že se odsazení současného řádku liší od předaného odsazení předchozího řádku, je volání funkce ukončeno. Pro každou neprimitivní proměnnou se

vytváří reference - jedinečný identifikátor, podle které lze pak dohledat přidružené proměnné. V praxi to znamená, že po rozkliknutí šipky u proměnné viz obrázek se zašle požadavek o proměnné s parametrem identifikátoru dané proměnné. Následně se rekurzivně proiterují úplně všechny načtené proměnné a vrátí se vnořenné proměnné v odpovědi na požadavek.

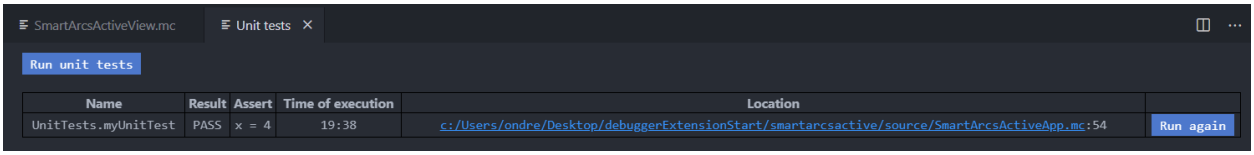
Příkaz `backtrace` poskytuje informaci o aktivních snímcích zásobníku v jakémkoliv časovém bodě během vykonávání programu. Výstup tohoto příkazu se zpracuje a zašle jako odpověď na žádost o stopu zásobníku (stack trace). Mezi jednotlivými snímky lze pak přepínat viz 4.6.



Obrázek 4.6: Zásobník volání

4.4 Unit testy

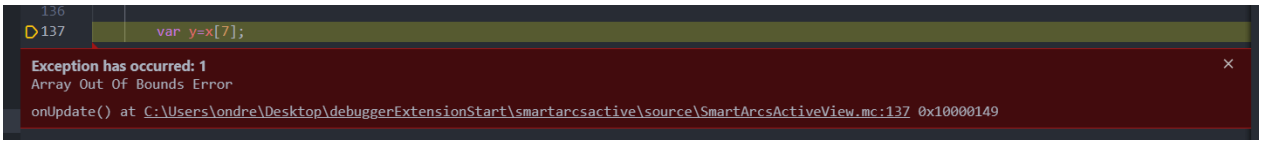
Rozšíření poskytuje grafické rozhraní k spouštění a vyhodnocování unit testů. Unit testy musí být umístěny v zdrojovém souboru s příponou .mc ve složce source, která v rámci projektu obsahuje všechny zdrojové soubory aplikace.



Obrázek 4.7: Unit testy

Grafické rozhraní unit testů je realizováno pomocí Webview viz obrázek 4.6. Po zmáčknutí tlačítka se do rozšíření zašle zpráva, která značí spuštění všech unit testů. Poté se zkompiluje projekt s příznakem -t, spustí simulátor a monkeydo skript ze složky Connect IQ SDK. Následuje vyhledání všech unit testů ve zdrojových souborech. Toto vyhledávání zprostředkovává npm balíček glob, který umožňuje hledat shody v souborech pomocí zadaného vzoru podobně jako je tomu u interpreta příkazů [18]. Glob prohledá rekurzivně všechny soubory ve složce source a v nich se pokusí najít unit testy. Informace o souboru a řádku, na kterém se jednotlivé unit testy nachází, se zašle do webview a zobrazí u každého unit testu v sloupci Location.

Rozšíření také umí reagovat na situace, kdy se program ukončí neošetřenou výjimkou či spadne. Pád může být například způsoben neošetřenou výjimkou 4.8, kdy dojde k přístupu k poli mimo jeho rozsah.



Obrázek 4.8: Neošetřená výjimka

Kapitola 5

Závěr a zhodnocení dosažených výsledků

Cílem této bakalářské práce bylo vytvořit plnohodnotné rozšíření pro ladění Connect IQ aplikací v editoru Visual Studio Code. Rozšíření je určené pro vývojáře Connect IQ aplikací, kteří byli doposud odkázáni na vývojové prostředí Eclipse, jelikož jenom tam bylo možné ladit Connect IQ aplikace.

Toto rozšíření poskytuje základní ladící funkcionality tak, jak je tomu ve vývojovém prostředí Eclipse. Pomocí tohoto rozšíření je možné programy vykonávat nebo zastavovat v určitých bodech a následně zkoumat vše, co se týká proměnných, volání funkcí a zásobníku. Před každým spuštěním programu se vybere zařízení, které je pak simulováno v Connect IQ simulátoru počas běhu programu.

Nabízí se také testování jednotlivých částí aplikace pomocí unit testů, jež je možné spouštět pomocí jednoduchého grafického rozhraní. V rámci rozšíření lze také vytvořit úplně nový typ projektu - Connect IQ aplikace a následně ji začít vyvíjet.

Literatura

1. *Connect IQ* [online]. Schaffhausen, Switzerland: Garmin Ltd., 2021 [cit. 2021-04-03]. Dostupné z: https://developer.garmin.com/connect-iq/.
2. *Monkey C* [online]. Schaffhausen, Switzerland: Garmin Ltd., 2021 [cit. 2021-04-03]. Dostupné z: https://developer.garmin.com/connect-iq/monkey-c/.
3. *Connect IQ Basics: Getting Started* [online]. Schaffhausen, Switzerland: Garmin Ltd., 2021 [cit. 2021-04-04]. Dostupné z: https://developer.garmin.com/connect-iq/connect-iq-basics/getting-started/.
4. *Core Topics: Debugging* [online]. Schaffhausen, Switzerland: Garmin Ltd., 2021 [cit. 2021-04-04]. Dostupné z: https://developer.garmin.com/connect-iq/core-topics/debugging/.
5. *Reference Guides: Jungle Reference* [online]. Schaffhausen, Switzerland: Garmin Ltd., 2021 [cit. 2021-04-04]. Dostupné z: https://developer.garmin.com/connect-iq/reference-guides/jungle-reference/.
6. *Core Topics: Unit Testing* [online]. Schaffhausen, Switzerland: Garmin Ltd., 2021 [cit. 2021-04-06]. Dostupné z: https://developer.garmin.com/connect-iq/core-topics/unit-testing/.
7. *Extensions Capabilities Overview* [online]. Seattle: Microsoft, 2021 [cit. 2021-04-07]. Dostupné z: https://code.visualstudio.com/api/extension-capabilities/overview.
8. *TypeScript Programming with Visual Studio Code* [online]. Seattle: Microsoft, 2021 [cit. 2021-04-07]. Dostupné z: https://code.visualstudio.com/docs/languages/typescript.
9. *VS Code API* [online]. Seattle: Microsoft, 2021 [cit. 2021-04-07]. Dostupné z: https://code.visualstudio.com/api/references/vscode-api.
10. *Overview* [online]. Seattle: Microsoft, 2021 [cit. 2021-04-07]. Dostupné z: https://microsoft.github.io/debug-adapter-protocol/overview.
11. *Run JavaScript Everywhere* [online]. San Francisco, California: OpenJS Foundation, 2021 [cit. 2021-04-07]. Dostupné z: https://nodejs.dev/.
12. *Debugger Extension* [online]. Seattle: Microsoft, 2021 [cit. 2021-04-07]. Dostupné z: https://code.visualstudio.com/api/extension-guides/debugger-extension.

13. *Node.js v15.14.0 Documentation: Child process* [online]. San Francisco, California: OpenJS Foundation, 2021 [cit. 2021-04-07]. Dostupné z: `https://nodejs.org/api/child_process.html`.

14. *Promise* [online]. San Francisco: Mozilla, 2021 [cit. 2021-04-10]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise`.

15. *Node.js v15.14.0 Documentation: Stream* [online]. San Francisco, California: OpenJS Foundation, 2021 [cit. 2021-04-10]. Dostupné z: `https://nodejs.org/api/stream.html`.

16. *Webview* [online]. Seattle: Microsoft, 2021 [cit. 2021-04-10]. Dostupné z: `https://code.visualstudio.com/api/extension-guides/webview`.

17. *Debugging* [online]. Seattle: Microsoft, 2021 [cit. 2021-04-10]. Dostupné z: `https://code.visualstudio.com/docs/editor/debugging`.

18. *Glob* [online]. Oakland, California: Npm, Inc., 2021 [cit. 2021-04-10]. Dostupné z: `https://www.npmjs.com/package/glob`.

19. CHERNY, Boris. *Programming TypeScript: Making Your JavaScript Applications Scale*. 1st Edition. O'Reilly Media, Inc., 2019. ISBN 978-1492037651.

20. JOHNSON, Bruce A. *Visual Studio Code: End-to-End Editing and Debugging Tools for Web Developers*. 1st Edition. John Wiley & Sons, 2019. ISBN 1119588189.

21. JEPSON, Brian. *Wearable Programming for the Active Lifestyle*. 1st Edition. O'Reilly Media, Inc., 2017. ISBN 9781491972090.